# 对象状态与一级缓存

## 状态介绍

* hibernate 规定三种状态：瞬时态、持久态、脱管态
* 状态

瞬时态：transient，session没有缓存对象，数据库也没有对应记录。

OID特点：没有值

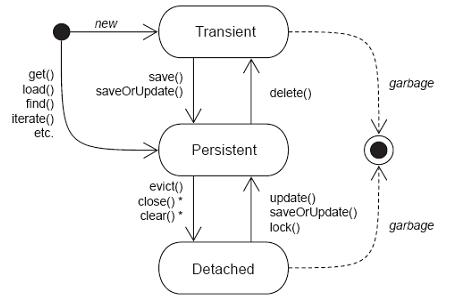
持久态：persistent，session缓存对象，数据库最终会有记录。（事务没有提交）

OID特点：有值

脱管态：detached，session没有缓存对象，数据库有记录。

OID特点：有值

## 转换



### 瞬时态/临时态

* 获得：一般都只直接创建（new）
* 瞬时态 转换 持久态

一般操作：save方法、saveOrUpdate

* 瞬时态 转换 脱管态

一般操作：通过setId方法设置数据

例如：

User user = new User(); //瞬时态

user.setUid(1); //脱管态

### 持久态

* 获得：

查询操作：get、loat、createQuery、createCriteria 等 获得都是持久态【】

执行save之后持久态

执行update之后持久态

* 持久态 转换 瞬时态

官方规定执行delete() --民间：删除态

* 持久态 转换 脱管态

session没有记录

session.close () 关闭

session.clear() 清除所有

session.evict(obj) 清除指定的PO对象

### 脱管态/游离态

* 获得：

创建、并设置OID的

通过api获得

* 脱管态 转换 瞬时态

手动去除OID，设置成默认值

* 脱管态 转换 持久态

一般操作：update（）、saveOrUpdate

|  |
| --- |
| @Test  **public** **void** demo01(){  User user = **new** User(); //瞬时态  user.setUsername("jack");  user.setPassword("1234"); //瞬时态（与oid没有关系）    Session session = factory.openSession();  session.beginTransaction();    session.save(user); //持久态  //---- 持久态就应该有持久态的行为（特性）    // user.setUsername("rose"); //持久态对象 被修改后，hibernate将自动生成update语句  // session.flush();      session.getTransaction().commit();  session.close();    System.*out*.println(user); //脱管态  } |

# 一级缓存

## 介绍

* 一级缓存：又称为session级别的缓存。当获得一次会话（session），hibernate在session中创建多个集合（map），用于存放操作数据（PO对象），为程序优化服务，如果之后需要相应的数据，hibernate优先从session缓存中获取，如果有就使用；如果没有再查询数据库。当session关闭时，一级缓存销毁。

## 一级缓存操作

### 证明一级缓存

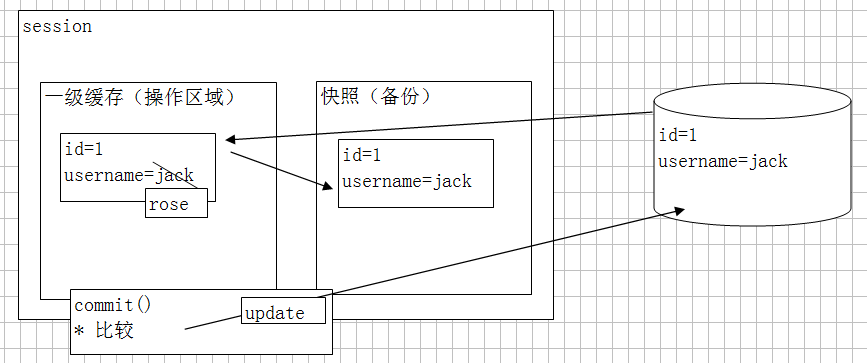
|  |
| --- |
| @Test  **public** **void** demo02(){  //证明一级缓存  Session session = factory.openSession();  session.beginTransaction();    //1 查询 id = 1  User user = (User) session.get(User.**class**, 1);  System.*out*.println(user);  //2 再查询 -- 不执行select语句，将从一级缓存获得  User user2 = (User) session.get(User.**class**, 1);  System.*out*.println(user2);    session.getTransaction().commit();  session.close();  } |

### 移除

|  |
| --- |
| @Test  **public** **void** demo03(){  //清除缓存  Session session = factory.openSession();  session.beginTransaction();    User user = (User) session.get(User.**class**, 1); //--select  System.*out*.println(user);    //清除  //session.clear();  session.evict(user);    // 一级缓存没有缓存对象，从数据库直接查询  User user2 = (User) session.get(User.**class**, 1); //--select  System.*out*.println(user2);    session.getTransaction().commit();  session.close();  } |

### 一级缓存快照【掌握】

* 快照：与一级缓存一样的存放位置，对一级缓存数据备份。保证数据库的数据与 一级缓存的数据必须一致。如果一级缓存修改了，在执行commit提交时，将自动刷新一级缓存，执行update语句，将一级缓存的数据更新到数据库。



### refresh 刷新

* refresh 保证 一级缓存的数据 与 数据库的数据 保持一致。将执行select语句查询数据库，将一级缓存中的数据覆盖掉。只要执行refresh都将执行select语句。

|  |
| --- |
| @Test  **public** **void** demo04(){  //刷新  Session session = factory.openSession();  session.beginTransaction();    User user = (User) session.get(User.**class**, 1); //--select  System.*out*.println(user);    session.refresh(user);    session.getTransaction().commit();  session.close();  } |

### 快照演示（一级缓存刷新）

|  |
| --- |
| @Test  **public** **void** demo05(){  //快照  Session session = factory.openSession();  session.beginTransaction();    User user = (User) session.get(User.**class**, 1); //--select  System.*out*.println(user);    //修改持久态对象内容（一级缓存内容）--默认在commit时，将触发update语句。  user.setUsername("rose2");      session.getTransaction().commit();  session.close();  } |

* 问题：一级缓存什么时候刷新？（了解）

默认情况提交（commit()）刷新。

|  |
| --- |
| @Test  **public** **void** demo06(){  //设置刷新时机  Session session = factory.openSession();  session.beginTransaction();    //1 设置  session.setFlushMode(FlushMode.*MANUAL*);    User user = (User) session.get(User.**class**, 1);  user.setUsername("rose4");    //1 查询所有 -- AUTO , 查询之前先更新，保存一级缓存和数据库一样的  //List<User> allUser = session.createQuery("from User").list();    //2手动刷新 --MANUAL 将执行update，注意：一级缓存必须修改后的  session.flush();    // 如果MANUAL 在执行commit 不进行update  session.getTransaction().commit();  session.close();  } |

## PO对象操作

### save & persist

* save方法：瞬时态 转换 持久态 ,会初始化OID

1.执行save方法，立即触发insert语句，从数据库获得主键的值（OID值）

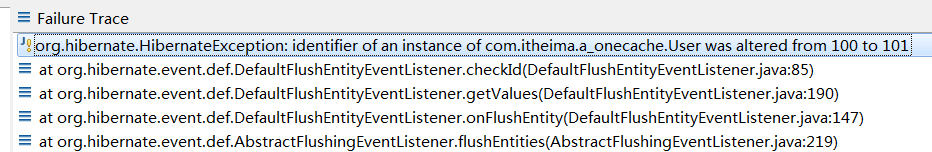
2.执行save方法前，设置OID将忽略。

3.如果执行查询，session缓存移除了，在执行save方法，将执行insert

|  |
| --- |
| @Test  **public** **void** demo01(){  User user = **new** User();  user.setUid(100);  user.setUsername("jack");  user.setPassword("1234");    Session session = factory.openSession();  session.beginTransaction();      session.save(user);      session.getTransaction().commit();  session.close();  } |

|  |
| --- |
| @Test  **public** **void** demo03(){  //代理 assigned  User user = **new** User();  //user.setUid(100);  user.setUsername("jack");  user.setPassword("1234");    Session session = factory.openSession();  session.beginTransaction();      session.save(user);      session.getTransaction().commit();  session.close();  } |

* 注意：持久态对象不能修改OID的值



|  |
| --- |
| @Test  **public** **void** demo04(){    Session session = factory.openSession();  session.beginTransaction();      User user = (User) session.get(User.**class**, 100);  user.setUid(101);    session.save(user);    session.getTransaction().commit();  session.close();  } |

* persist方法：瞬时态 转换 持久态 ,不会立即初始化OID

注意： persist方法不会立即得到ID,所以执行sql语句的时机要靠后.

### update

* update：脱管态 转换 持久态

如果OID在数据存放的，将执行update语句

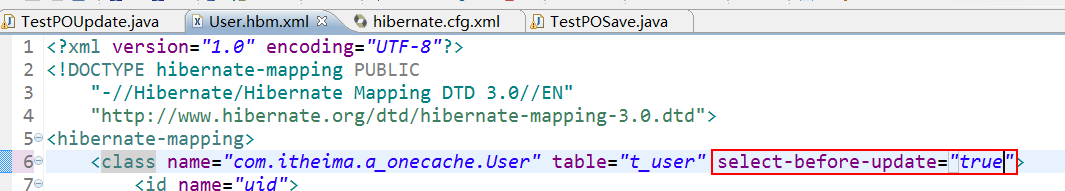
如果OID不存在将抛异常

|  |
| --- |
| @Test  **public** **void** demo01(){  //自然 assigned  User user = **new** User();  user.setUid(101);  user.setUsername("jack1");  user.setPassword("12345");    Session session = factory.openSession();  session.beginTransaction();      session.update(user);      session.getTransaction().commit();  session.close();  } |

* 注意1：如果数据没有修改，执行save方法，将触发update语句。

查询速度 比 更新速度快

通过<class select-before-update> 来设置更新前先查询，如果没有改变就不更新。



总结：

update 之后对象 持久态

|  |
| --- |
| @Test  **public** **void** demo03(){  // merge 合并  User user = **new** User();  user.setUid(1);  user.setUsername("jack3");  user.setPassword("12345");    Session session = factory.openSession();  session.beginTransaction();      // 1 oid =1 持久态对象  User user2 = (User) session.get(User.**class**, 1);    // session.update(user);  session.merge(user);      session.getTransaction().commit();  session.close();  } |

### saveOrUpdate

* 代理主键：

判断是否有OID

如果没有OID，将执行insert语句

如果有OID，将执行update语句。

|  |
| --- |
| @Test  **public** **void** demo02(){  // 代理 native  User user = **new** User();  // user.setUid(2);  user.setUsername("jack2");  user.setPassword("12345");    Session session = factory.openSession();  session.beginTransaction();      session.saveOrUpdate(user);      session.getTransaction().commit();  session.close();  } |

* 自然主键：

先执行select语句，查询是否存放

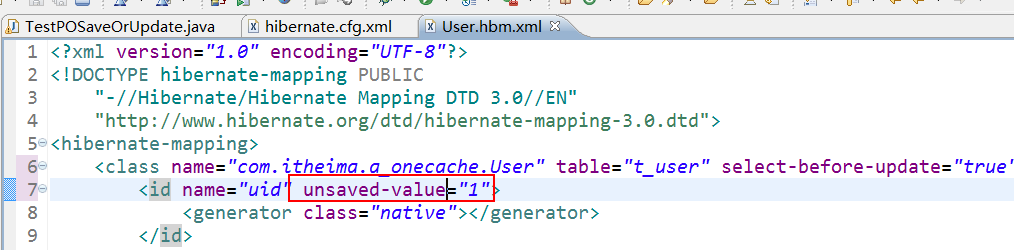
如果不存在，将执行insert

如果存在，将执行update

|  |
| --- |
| @Test  **public** **void** demo02(){  // 自然 assigned  User user = **new** User();  user.setUid(2);  user.setUsername("jack2333");  user.setPassword("12345333");    Session session = factory.openSession();  session.beginTransaction();      session.saveOrUpdate(user);      session.getTransaction().commit();  session.close();  } |

* 注意1：native下，默认OID是否存在，使用默认值。例如：Integer 默认null

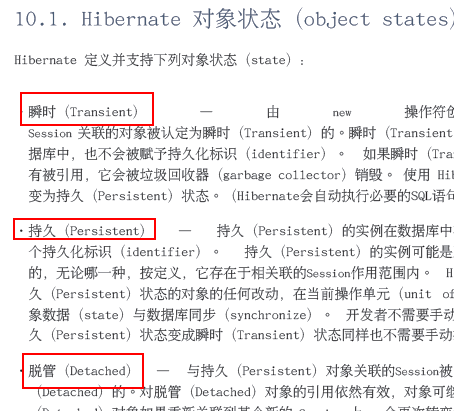
通过<id unsaved-value="1"> 修改使用默认值，如果设置1进行insert语句。此内容提供hibernate使用的，录入到数据库后，采用自动增长。



### delete

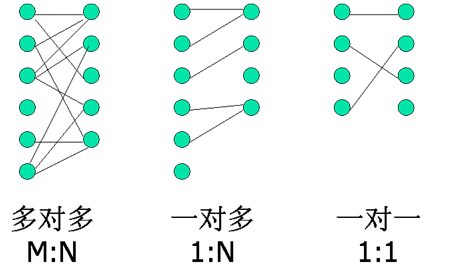
总结：

PO对象状态：瞬时态、持久态、脱管态



# 多表设计

* 在开发中，前期进行需求分析，需求分析提供E--R图，根据ER图编写表结构。
* 表之间关系存在3种：一对多、多对多、一对一。（回顾）



一对多：1表（主表）必须主键 和 多表（从表）必须外键，主表的主键 与 从表外键 形成主外键关系

多对多：提供中间表（从表），提供2个字段（外键）分别对应两个主表。

一对一：？？？

* 面单对象描述 对象与对象 之间关系？【掌握】

一对多：客户和订单

private class Customer{

//一对多：一个客户 拥有 多个订单

private Set<Order> orderSet;

}

private class Order{

//多对一：多个订单 属于 一个客户

private Customer customer;

}

多对多：Student学生 和 Course课程

private class Student{

//多对多：多个学生（当前）学习 【不同课程】

private Set<Course> courseSet ...;

}

private class Course{

//多对多：多个课程 可以被 【不同学生】学习

private Set<Student> student = ...;

}

一对一：公司company 和 地址address

private class Company{

private Address address;

}

private class Address{

private Company company;

}

# 关联关系映射

## 一对多实现【掌握】

### 实现类

|  |
| --- |
| **public** **class** Customer {    **private** Integer cid;  **private** String cname;    //一对多：一个客户（当前客户） 拥有 【多个订单】  // \* 需要容器存放多个值，一般建议Set （不重复、无序）  // \* 参考集合：List、Map、Array等  // \*\* 建议实例化--使用方便  **private** Set<Order> orderSet = **new** HashSet<Order>(); |

|  |
| --- |
| **public** **class** Order {  **private** Integer xid;  **private** String price;    //多对一：多个订单属于【一个客户】  **private** Customer customer; |

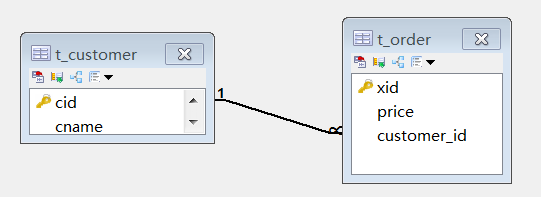
### 配置文件

* Customer.hbm.xml

|  |
| --- |
| <class name=*"com.itheima.b\_onetomany.Customer"* table=*"t\_customer"*>  <id name=*"cid"*>  <generator class=*"native"*></generator>  </id>  <property name=*"cname"*></property>    <!-- 一对多：一个客户（当前客户） 拥有 【多个订单】  1 确定容器 set <set>  2 name确定对象属性名  3 确定从表外键的名称  4 确定关系，及另一个对象的类型  注意：  在hibernate中可以只进行单向配置  每一个配置项都可以完整的描述彼此关系。  一般情况采用双向配置，双方都可以完成描述表与表之间关系。  -->  <!-- 一对多：一个客户（当前客户） 拥有 【多个订单】 -->  <set name=*"orderSet"* cascade=*"delete-orphan"*>  <key column=*"customer\_id"*></key>  <one-to-many class=*"com.itheima.b\_onetomany.Order"*/>  </set>  </class> |

* Order.hbm.xml

|  |
| --- |
| <class name=*"com.itheima.b\_onetomany.Order"* table=*"t\_order"*>  <id name=*"xid"*>  <generator class=*"native"*></generator>  </id>  <property name=*"price"*></property>    <!-- 多对一：多个订单属于【一个客户】  \* name 确定属性名称  \* class 确定自定义类型  \* column 确定从表的外键名称  -->  <many-to-one name=*"customer"* class=*"com.itheima.b\_onetomany.Customer"* column=*"customer\_id"*></many-to-one>    </class> |



## 一对多操作

### 保存客户

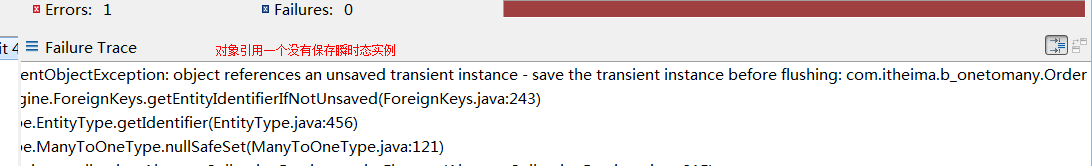
|  |
| --- |
| @Test  **public** **void** demo01(){  // 1 创建客户，并保存客户--成功  Session session = factory.openSession();  session.beginTransaction();    Customer customer = **new** Customer();  customer.setCname("田志成");    session.save(customer);    session.getTransaction().commit();  session.close();  } |

### 保存订单

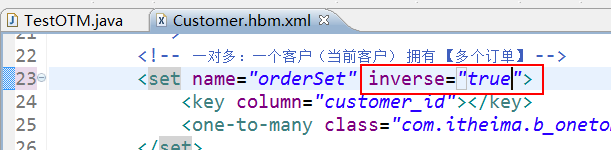
|  |
| --- |
| @Test  **public** **void** demo02(){  // 2 创建订单，保存订单--成功，外键为null  Session session = factory.openSession();  session.beginTransaction();    Order order = **new** Order();  order.setPrice("998");    session.save(order);    session.getTransaction().commit();  session.close();  } |

### 客户关联订单，只保存客户

|  |
| --- |
| @Test  **public** **void** demo03(){  // 3 创建客户和订单，客户关联订单，保存客户？  Session session = factory.openSession();  session.beginTransaction();    //1 客户和订单  Customer customer = **new** Customer();  customer.setCname("成成");    Order order = **new** Order();  order.setPrice("998");    //2 客户关联订单  customer.getOrderSet().add(order);    //3 保存客户  session.save(customer);    session.getTransaction().commit();  session.close();  } |



### 双向关联，使用inverse

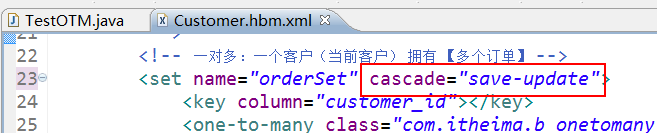


|  |
| --- |
| @Test  **public** **void** demo04(){  // 4 创建客户和订单，客户关联订单，订单也关联客户，保存客户和订单？  // \* 开发中优化程序 ， n + 1 问题？  // \*\* 解决方案1：客户不关联订单 ,不建议  // \*\* 解决方案2：客户放弃对订单表外键值的维护。  // \*\*\*\* Customer.hbm.xml <set name="orderSet" inverse="true">  // \*\* inverse 将维护外键值的权利交予对象。相当于自己放弃。（反转）  Session session = factory.openSession();  session.beginTransaction();    //1 客户和订单  Customer customer = **new** Customer();  customer.setCname("成成");    Order order = **new** Order();  order.setPrice("998");    //2 客户关联订单  customer.getOrderSet().add(order);  //3 订单也关联客户  order.setCustomer(customer);    //4 保存客户  // \* 1 save(order) -- insert --> 1,998 null  // \* 2 订单管理客户，此时null --预留update --> 更新所有（正常设置）  // \* 3 save(customer) -- insert --> 1,成成  // \* 4 客户关联订单 --> 预留update --> 更新订单外键 （维护外键）  // \* 5 提交commit --> 执行2 和 4  session.save(order);  session.save(customer);    session.getTransaction().commit();  session.close();  } |

* 在一对多开发中，一方一般都放弃对外键值的维护。及<set inverse="true

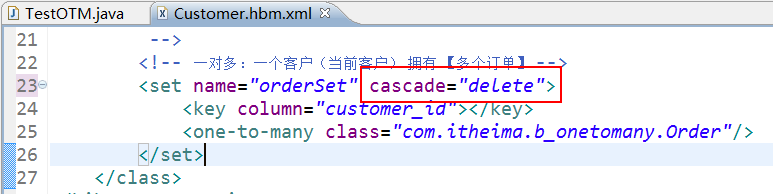
## 级联操作（读、理解）

### save-update 级联保存或更新



|  |
| --- |
| @Test  **public** **void** demo032(){  // 32 创建客户和订单，客户关联订单，保存客户？ --抛异常  // \*\* 解决方案2：级联操作--级联保存或更新  // \*\* Customer.hbm.xml <set cascade="save-update">  // \*\* 在保存客户的同时，一并保存订单  Session session = factory.openSession();  session.beginTransaction();    //1 客户和订单  Customer customer = **new** Customer(); //瞬时态  customer.setCname("成成");    Order order = **new** Order(); //瞬时态  order.setPrice("998");    //2 客户关联订单  customer.getOrderSet().add(order);    //3 保存客户  session.save(customer); //持久态  // 关联操作都是持久态的，此时 持久态Customer 引用 一个 瞬时态的Order 抛异常    session.getTransaction().commit();  session.close();  } |

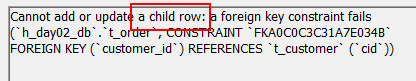
### delete 级联删除

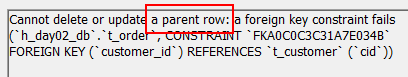


|  |
| --- |
| @Test  **public** **void** demo05(){  // 5 查询客户，并删除(持久态)  // 默认：当删除客户，默认将订单外键设置成null。  // 级联删除：删除客户时，并将客户的订单删除。  // \*\* Customer.hbm.xml <set name="orderSet" cascade="delete">  Session session = factory.openSession();  session.beginTransaction();    Customer customer = (Customer) session.get(Customer.**class**, 10);    session.delete(customer);      session.getTransaction().commit();  session.close();  } |

### 孤儿删除

* 一对多关系，存在父子关系。1表（主表）可以成为父表，多表（从表）也可以子表。

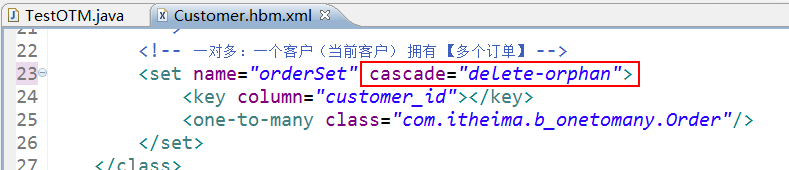




总结：

主表不能删除，从表已经引用（关联）的数据

从表不能添加，主表不存在的数据。



|  |
| --- |
| @Test  **public** **void** demo06(){  // 6 查询客户，并查询订单，解除客户和订单订单的关系  // \* 默认：客户和订单解除关系后，外键被设置成null，此时订单就是孤儿。客户和订单都存在。  // \* 孤儿删除（孤子删除），当订单称为孤儿，一并删除。客户仍存在。  Session session = factory.openSession();  session.beginTransaction();    //1 查询客户  Customer customer = (Customer) session.get(Customer.**class**, 9);    //2查询订单  Order order = (Order) session.get(Order.**class**, 8);    //3 解除关系  customer.getOrderSet().remove(order);    session.getTransaction().commit();  session.close();  } |

### 总结

save-update：A保存，同时保存B

delete：删除A，同时删除B，AB都不存在

delete-orphan：孤儿删除，解除关系，同时将B删除，A存在的。

如果需要配置多项，使用逗号分隔。<set cascade="save-update,delete">

all : save-update 和 delete 整合

all-delete-orphan : 三个整合